ECAI 2014
T. Schaub et al. (Eds.)

doi:10.3233/978-1-61499-419-0-1027

Generation of Relevant Spreadsheet Repair Candidates

Birgit Hofer[1] and **Rui Abreu**[2] and **Alexandre Perez**[2] and **Franz Wotawa**[1]

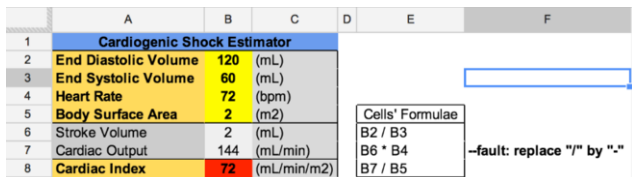
Abstract. Spreadsheets are amongst the most successful examples of end user programming. Because of their, still increasing, importance for companies, spreadsheets have drastic economical and societal impact. Hence, locating and fixing spreadsheet faults is important and deserves attention from the research community. A state-of-the-art technique uses genetic programming for generating repair candidates, but a limitation that hinders real-world application is that it still computes too many repair candidates. In this paper, we discuss a novel technique based on constraint solving that uses distinguishing test cases to narrow down the number of repair candidates.

1 Introduction

Spreadsheets can be regarded as a highly flexible end-users programming environment [11]. These so-called "end-user" programmers vastly outnumber professional ones: the US Bureau of Labor and Statistics estimates that, in 2012, more than 55 million people used spreadsheets and databases at work on a daily basis [11]. However, numerous studies have shown that existing spreadsheets contain errors at an alarmingly high rate, e.g., [4]. These errors may entail a serious economical impact for the business, causing yearly losses worth roughly 10 billion dollars [13].

Several researchers have developed methods and techniques for improving the overall quality of spreadsheets, e.g., [1, 3, 5, 7, 8, 10]. Approaches which not only localize potential faulty cells but also suggest a repair (e.g., [1, 9]) are particularly interesting. Genetic programming [9] generates repair suggestions from a faulty spreadsheet and a failing test case[3] by mutating randomly chosen formula cells. If the created mutant passes the given test case, a potential repair candidate is found. Unfortunately, such approaches often generate too many repair candidates. A large number of repair candidates often overwhelms the user. To avoid this problem, we propose in this paper the MUSSCO (Mutation Supported Spreadsheet COrrection) approach which filters out wrong repair candidates by using distinguishing test cases. A test case is a distinguishing test [12] case if and only if there is at least one output cell where the computed value of two mutated versions of a spreadsheet differ on the same input.

We use the running example in Figure 1 to illustrate our approach. Physicians use this spreadsheet to estimate cardiogenic shock. Cells `B2` to `B5` need an input from the user. Cell `B8` shows the result of the computation from which physicians derive their conclusions. Cell `B6` is faulty because it computes `B2/B3` instead of `B2-B3`. The test case T ($I = \{\ell(\texttt{B2}) = 120, \ell(\texttt{B3}) = 60, \ell(\texttt{B4}) = 72, \ell(\texttt{B5}) = 2\}, O =$



	A	B	C	D	E	F
1	Cardiogenic Shock Estimator					
2	End Diastolic Volume	120	(mL)			
3	End Systolic Volume	60	(mL)			
4	Heart Rate	72	(bpm)			
5	Body Surface Area	2	(m2)			
6	Stroke Volume	2	(mL)		Cells' Formulae	
7	Cardiac Output	144	(mL/min)		B2 / B3	
8	Cardiac Index	72	(mL/min/m2)		B6 * B4	--fault: replace "/" by "-"
					B7 / B5	

Figure 1. The Cardiogenic shock estimator spreadsheet

$\{\ell(\texttt{B8}) = 2160\}$), where $\ell(\texttt{c})$ represents the value of cell `c`, is a failing test case because the computed value for `B8` (72) differs from the expected value (2160). A debugging approach which generates repair candidates (e.g. [1, 9]) would, for example, return changing cell `B6` to `B2 - B3` (mutant Π_1) or changing `B7` to `30 * B6 * B4` (mutant Π_2) as repair candidates because both make the test case T a passing one. In contrast, MUSSCO would first generate a distinguishing test case for Π_1 and Π_2 so that it yields different output values for both mutants, e.g.: $\ell(\texttt{B2}) = 30, \ell(\texttt{B3}) = 30, \ell(\texttt{B4}) = 30, \ell(\texttt{B5}) = 1$. Afterwards, MUSSCO asks the user which output is correct and discards all mutants which fail this new test case.

2 Computing distinguishing test cases

For computing distinguishing test cases, we convert mutants into a constraint satisfaction problem (CSP) [6]. A CSP is a tuple (V, D, C) where V is a set of variables with a corresponding domain from D, and C is a set of constraints. Each constraint has a set of variables, i.e., its scope, and specifies the relation between the variables. A solution of a CSP is an assignment of values to variables such that all constraints are fulfilled. Algorithm 1 describes the creation of distinguishing test cases. This algorithm takes as input two mutated versions of the same spreadsheet. In lines 1 and 2, the functions GETINPUTCELLS and GETOUTPUTCELLS are called. These functions return the set of input and output cells for the given spreadsheet. An input cell is a cell that does not reference another cell. Conversely, an output cell is a cell that is not referenced by another cell. In lines 3 and 4, the mutants are converted into their constraint representation. This conversion is based on the conversion explained by Abreu et al. [2][4]. The second parameter of the function CONVERT is a constant that acts as postfix for variables. This postfix is necessary to distinguish the constraint representation of m_1 from that of m_2: Each variable in the constraint system for mutant m_1 gets the postfix "_1", each variable for mutant m_2 gets the postfix "_2". In line 5, a constraint is created that ensures that the input of m_1 is equal to the input of m_2. In line 6, a constraint is created that ensures that at least one output cell of m_1 has a different value than the same output cell in m_2. The function GETSOLUTION calls the constraint solver with

[1] Graz University of Technology, Austria, bhofer@ist.tugraz.at, wotawa@ist.tugraz.at
[2] University of Porto, Portugal, rui@computer.org, alexandre.perez@fe.up.pt
[3] A test case is a tuple (I, O), where I are the values for the input cells and O the expected values for other cells. A test case is a passing test case if all computed values are equivalent to the expected values. Otherwise, it is a failing test case.
[4] [2] focuses on computing diagnoses (i.e. fault localization), while this work focuses on generating distinguishing test cases. Nevertheless, the conversion into constraints follows principally the same rules.

these constraints (line 8). This function either returns a distinguishing test case, UNSAT (in case of equivalent mutants) or UNKOWN (in case of undecidability).

Algorithm 1 GETDISTINGUISHINGTESTCASE(m_1, m_2)

Require: Mutants of a spreadsheet m_1, m_2
Ensure: A distinguishing test case or UNSAT/UNKOWN
1: inputCells = GETINPUTCELLS(m_1)
2: outputCells = GETOUTPUTCELLS(m_1)
3: Cons1 =CONVERT($m_1 \setminus$ inputCells, "_1")
4: Cons2 =CONVERT($m_2 \setminus$ inputCells, "_2")
5: inputCon = $\bigwedge_{c \in inputCells} c\_1 = c\_2$
6: outputCon = $\bigvee_{c \in outputCells} c\_1 \neq c\_2$
7: Cons = Cons1 $\cup$ Cons2 $\cup$ inputCon $\cup$ outputCon
8: **return** GETSOLUTION(Cons)

With the automatic generation of distinguishing test cases, we are now able to narrow down the number of repair candidates that are presented to the user. MUSSCO takes as input a set of possible repair candidates (mutants). A primitive way to compute mutants is to clone the spreadsheet and change arbitrary operators and operands in all formulas of the cells contained in one diagnosis (diagnoses could be obtained using the approach in [2]). If the created mutant satisfies the given test case the mutant is presented to the user, otherwise it is discarded. The problem with this approach is that too many mutants have to be computed until the first mutant passes the given test case. Therefore, a more sophisticated approach which includes the mutation creation process in the CSP is used. Instead of only transforming cell formulas into a value-based constraint model, we also include the information how the cells could be mutated. We are aware that our approach could not generate mutants for all types of faults, and a generalization remains for future work.

Algorithm 2 describes the repair filtering phase of our approach, which is invoked after the set of repair candidates M (mutants) has been generated. In lines 1 and 2, the sets eqMut and undesMut, used to store the pairs of equivalent and undecidable mutants, are initialized. If M contains at least two mutants which

Algorithm 2 Algorithm MUSSCO(M)

Require: A set of repair candidates M
Ensure: A set of possible corrections
1: eqMut =$\emptyset$
2: undesMut =$\emptyset$
3: **while** $|M| \geq 2 \wedge \exists((m_1, m_2) \in M : (m_1, m_2) \notin$ eqMut $\wedge$ $(m_1, m_2) \notin$ undesMut) **do**
4: Select two mutants m_1, m_2 from M where $(m_1, m_2) \notin$ eqMut $\wedge$ $(m_1, m_2) \notin$ undesMut
5: $T' =$ GETDISTINGUISHINGTESTCASE(m_1, m_2)
6: **if** $T' =$ UNSAT **then**
7: eqMut = eqMut $\cup \{(m_1, m_2)\}$
8: **else**
9: **if** $T' =$ UNKNOWN **then**
10: undesMut = undesMut $\cup \{(m_1, m_2)\}$
11: **else**
12: $T' = T' \cup$ GETEXPECTEDOUTPUT(T)
13: $M =$ FILTER(T', M)
14: **end if**
15: **end if**
16: **end while**
17: **return** M

are not equivalent or undecidable, such a pair of mutants is selected (line 4). In line 5, we call the test case retrieval function GETDISTINGUISHINGTESTCASE. If this function returns UNSAT, the pair m_1, m_2 is added to the set eqMut (line 7). If the function returns UNKNOWN, the pair m_1, m_2 is added to the set undesMut (line 10). Otherwise, the function returns a new test case. The function GETEXPECTEDOUTPUT is used to determine the expected output for the given test case (line 12). This function either asks the user or another oracle, e.g. a correct implementation of the spreadsheet. The function FILTER checks which mutants in M pass the new test case and returns these mutants (line 13).

3 Conclusions

The number of repair candidates produced by current debugging techniques for spreadsheets often overwhelms the user. To overcome this major drawback, we propose the MUSSCO approach which narrows down the number of repair candidates by using distinguishing test cases. We performed an initial case study on the publicly available Integer Spreadsheet Corpus (https://dl.dropbox.com/u/38372651/Spreadsheets/Integer_Spreadsheets.zip). The results of this preliminary evaluation show that on average 3.1 distinguishing test cases are generated and 3.2 mutants are reported as possible fixes. On average, the generation of the mutants and distinguishing test cases requires 47.9 seconds in total (on an Intel Core i7-3770K CPU and 16GB RAM). These results are promising as the required user interaction is low, but these results also indicate that further efforts should be spent to minimize the computation time.

REFERENCES

[1] Robin Abraham and Martin Erwig, 'GoalDebug: A spreadsheet debugger for end users', in *Proc. ICSE '07*, pp. 251–260, (2007).
[2] Rui Abreu, Birgit Hofer, Alexandre Perez, and Franz Wotawa, 'Using constraints to diagnose faulty spreadsheets', *Software Quality Journal*, 1–26, (2014).
[3] Yanif Ahmad, Tudor Antoniu, Sharon Goldwater, and Shriram Krishnamurthi, 'A type system for statically detecting spreadsheet errors', in *Proc. ASE '03*, pp. 174–183, (2003).
[4] David Chadwick, Brian Knight, and Kamalasen Rajalingham, 'Quality control in spreadsheets: A visual approach using color codings to reduce errors in formulae', *Software Quality Control*, **9**(2), 133–143, (2001).
[5] Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva, 'MDSheet: A framework for model-driven spreadsheet engineering', in *Proc. ICSE '12*, pp. 1395–1398, (2012).
[6] Rina Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.
[7] Felienne Hermans, Martin Pinzger, and Arie van Deursen, 'Detecting and visualizing inter-worksheet smells in spreadsheets', in *Proc. ICSE '12*, pp. 441–451, (2012).
[8] Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner, 'On the empirical evaluation of fault localization techniques for spreadsheets', in *Proc. FASE 2013*, (2013).
[9] Birgit Hofer and Franz Wotawa, 'Mutation-based spreadsheet debugging', in *Proc. IWPD '13*, pp. 132–137. IEEE, (2013).
[10] Dietmar Jannach and Ulrich Engler, 'Toward model-based debugging of spreadsheet programs', in *Proc. JCKBSE '10*, pp. 252–264, (2010).
[11] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck, 'The state of the art in end-user software engineering', *ACM Comput. Surv.*, **43**(3), 21:1–21:44, (April 2011).
[12] Mihai Nica, Simona Nica, and Franz Wotawa, 'On the use of mutations and testing for debugging', *Software : Practice & Experience*, (2012).
[13] Raymond R. Panko, 'Applying code inspection to spreadsheet testing', *J. of Management Information Systems*, **16**, 159–176, (Sept. 1999).